#2. Semaphore & Condition Variable

다음 사항을 의사코드로 작성하여 리포트로 제출하시오.

1. Monitor (a lock & Condition variables)로 Semaphore를 구현하시오.
2. Semaphore로 Condition Variable을 구현 하시오.

우선 세마포어의 구조는 다음과 같습니다.

```
class SEMAPHORE {
      int   ball        // 조건문 확인을 위한 변수 추가
      func  P(ball)     // 주머니에서 공을 하나 뺌
      func  V(ball)     // 주머니에 공 하나를 넣음
}
```

다음 모니터의 구조는 다음과 같습니다.

```
class MONITOR {
      lock  Lock{
            Acquire() { pending_all_interrupts };
            Release() { not_pending_interrupts };
      }
      condition  c1, c2 ·····

      function   f1() { ····· }
      function   f2() { ····· }

            .......
}
```

위의 **세마포어를 아래의 모니터 구조로 구현(1)**하고자 합니다. 원래 세마포어는 내부의 실제 변숫값을 간접적으로밖에 알 수 없으나, 모니터를 이용한 구현 간 조건문 확인을 위해 ball이라는 count value를 추가하였습니다

세마포어에서 P(), V() 함수는 Atomic 하게 실행되어야 합니다. 모니터로 이를 구현하기 위해선 해당 함수가 실행될 때 반드시 Lock을 얻고 함수를 실행해야 할 것입니다.

공이 가득 차있을 때에는 V() 연산을, 반대로 공이 비었을 때에는 P() 연산을 해선 안 됩니다. 따라서 적어도 하나의 condition value를 두어 이런 일이 일어나지 못하게 해주어야 합니다.

함수 내에서 공 개수 조건을 확인할 때, Mesa-Style과 Horae-Style이 있으나 일반적으로 Mesa-Style을 사용하기에 이로 구현하였습니다. while loop 내부가 그리 복잡하지 않고, 조건에 맞는 스레드를 sleep 시켜 Wait Queue에 넣어주는 기능도 같이 적용되었기에, Busy-Waiting 문제는 크게 일어나지 않을 거라 생각하였습니다

따라서 다음 장과 같이 구현하였습니다.

```
class MY_SEM {

      int ball              // shared data for while condition
      lock Lock             // only one lock

      class condition {     // for prevent wrong cases

            wait(lock *Lock) {
                  Lock->Release & go to sleep() // Atomic
                  Lock->Acquire
            }

            signal(item){
                  put item on Ready Queue
            }
      }

      function P() {

            Lock->Acquire()

            while (ball == EMPTY) // Mesa
                  condition.wait(&Lock);

            ball = ball-1
            SEMAPHORE->P()  // Do what real P() do

            if (something V on Wait Queue)
                  signal( one V )

            lock->Release()
      }

      function V() {

            Lock->Acquire()

            while (ball == FULL) // Mesa
                  condition.wait(&Lock);

            ball = ball+1
            SEMAPHORE->V() // Do what real V() do

            if (something P on Wait Queue)
                  signal( one P )

            lock->Release()
      }

}
```

다음으로 **condition Value를 세마포어로 구현(2)**하고자 합니다. Wait 함수에는 Lock이 Release 되었다가 다시 Acquire 되는 과정이 있습니다. 이를 공 하나짜리 세마포어로써 Lock을 대체 해보았습니다.

기존의 세마포어는 다수의 공을 사용하여, 일종의 Counter의 역할도 하였지만, 여기서는 단순히 Mutual Execution을 위한 Flag의 역할만 하므로 그냥 'Mutex'라고 하였습니다.

기존의 컨디션 벨류에서 'Lock->Require'와 '진행중인 스레드를 Wait Queue에 넣어주는 행위'를 동시에 실행하게 해야 합니다. 뮤텍스의 isBallExists라는 실제 세마포어 외부의 플래그 변수를 이용하여 이를 가능하게 구현하였습니다.

```
class Mutex {

      bool isBallExists // True면 Lock->Acquire() 가 진행된 상태

      func  P() {       // 공 하나만 들어가는 주머니에서 공을 하나 뺌

            while (isBallExists == FALSE)
                  ;     // 공이 없으면 Loop을 반복

            SEMAPHORE->P(isBallExists) // 실제 세마포어 P() 함수

            isBallExists = FALSE & go to sleep()
                                       // 진행되던 스레드는 Sleep 상태로
      }

      func  V()   {     // 공 하나만 들어가는 주머니에 공 하나를 넣음

            SEMAPHORE->V(isBallExists) // 실제 세마포어 V() 함수

            isBallExists = TRUE   // 다른 스레드의 P()함수가 while Loop
                                  // 을 탈출해 Sleep 상태가 되어 멈추도록
                                  // 신호를 남긴다.
      }
}

class example_condition {

      wait() {
            Mutex.P()  // Lock->Release() & go to sleep() 를 대체
                       // : 진행중인 스레드를 WaitQueue에 넣어둠

            Mutex.V()  // Lock->Acquire() 를 대체
                       // : 다시 레디큐로 돌아와 스케줄러에 의해 진행된 스래드는
                       //   여기서 자신이 크리티컬 섹션에 들어왔다는 신호를 남김
      }

      signal(item) { put item on Ready Queue }

      broadcast() { put all items on Ready Queue }
}
```